# Edit, Run, Error, Repeat: Patterns of First Year Programming Students

Johan Snider
*Uppsala University*
Uppsala, Sweden
johan.snider@it.uu.se

***Abstract*—Watching a student program can be a joyous or painstaking experience. Some students are able to successfully implement programming concepts in code, while other students struggle immensely. What are the key differences between these students?**

**It is possible with log data from development environments to get insights into students' programming behaviors. In this project we propose a way to process log data from students programming environments and reason about how we can analyze students programming behavior over time.**

***Index Terms*—Computer Science Education, Student Programming Behavior**

## I. Context

Log data from programming environments can be used to gain insight into students' programming behavior [1]. Data such as: program edit events, program run events (or program executions), error events and snapshots of code, are all examples of what can be contained in log data. When these events are timestamped they are referred to as fine grain log data [2]. This fine grain log data can be recorded by the development environment either online in a web browser, or locally on a computer. Once recorded, this data can be used to recreate a student's programming activity, or visualize that activity in various ways [3].

Analysis of fine grain log data have been used for:

- predicting final course grades
- identifying common patterns of programming [4]
- finding the most common student errors [1]
- calculating how often students take breaks [2]

Most of these projects have followed students for between one to four weeks, and analyzed less than eight programming assignments. Less longitudinal work has been found following students over longer periods of time. Following a group of first year programming students over several months working on dozens of programming assignments would give a better picture of students behavior in the course, and also provide the opportunity to see how students behavior changes over time.

## II. Objective

Students in first year programming courses demonstrate varying levels of programming ability. Anecdotal evidence shows that some students learn new concepts and implement them into code with minimal effort, while other students struggle immensely to implement programming concepts into code. This leads to two possible research questions:

1) What strategies, behaviors and patterns do students who find programming easy have that students that struggle lack?
2) What intervention or teaching is required to transition a student who struggles into a student who flourishes?

To find answers to these larger questions we can ask a set of smaller measurable questions, which can hopefully serve as stepping stones to answer the larger questions. These questions are readily answered with fine grain log data:

1) How long do students actively program in class?
2) What is the ration between: edits, runs and errors?
3) What happens after a student encounters an error?
4) Which errors occur most over time?
5) How often and for how long do students take breaks?

## III. Method

For the analysis of fine course log data from programming, we propose a state machine which can generate a text string to visualize a students work. The state machine is shown in Figure 1. Here we define active programming as the transition between editing, running and handling errors. We would also have to set a break threshold to establish the difference between a student thinking in one state, and taking a break to do something else [2]. This break threshold could be as little as a few seconds, or set to several minutes depending on the context. Other program related tasks such as looking at documentation or going back to a previous exercise to see an example of code are not included in the figure.

This state machine can take fine grain log data as input and generate a string of text which can give insight into the students development process. These strings can be compared with other students working on the same exercise or assignment. Without automated analysis this could be a way for a teacher to get a quick overview of how a student has been working on an assignment. With automated analysis, we could provide teachers with insights which could be helpful to improve their teaching. We could also identify patterns students use for solving programming assignments. This could be especially helpful for students who struggle, if we can identify actions which can help the student learn to program more efficiently.

Supervisor Anna Eckerdahl

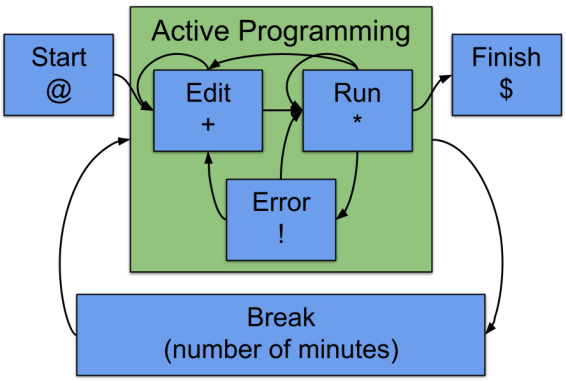


Fig. 1. Edit, Run, Error State Machine

This state machine itself can also be used as a tool to show students that programming is a process, which includes handling errors and debugging.

## IV. Results

Listing 1 shows a code example of what students could be expected to program to pass an exercise. In this example the code is logic for the game Rock, Paper, Scissors. At most there are nine conditions to check, but this can be reduced to five or even three cases. In the example, there are five cases. There are multiple ways to implement this code, and students will find different valid solutions for the exercise.

```
def drawWinner(leftMove, rightMove):
    # Check leftMove and rightMove
    if (leftThrow.value == rightThrow.value):
        Label('Tie', 200, 200, size=60)

    elif (leftThrow.value == 'Rock' and
          rightThrow.value != 'Paper'):
        Label('Win', 200, 200, size=60)
    elif (leftThrow.value == 'Scissors' and
          rightThrow.value != 'Rock'):
        Label('Win', 200, 200, size=60)
    elif (leftThrow.value == 'Paper' and
          rightThrow.value != 'Scissors'):
        Label('Win', 200, 200, size=60)

    else:
        Label('Lose', 200, 200, size=60)
```

Listing 1. Python Snippet

Looking at some synthesized data in Table I, we see different patterns which could be generated by students solving this exercise. We see that student A starts out editing and running their program, encounters a few errors, and then goes on to complete the assignment. Specifically we see that this student often runs their program after editing it. This could be indicative of a more methodical approach, where the student wants to make sure that their code changes work in the way they expect. Student B seems to encounter several errors at the beginning of their work, then continues to edit their program heavily and then run into more errors before deciding to stop working on the exercise. It could be the case here that student B does not have as developed programming strategies

| Student | Pattern |
|---|---|
| A | @+*+*++*++*+*!*!*+*!*!+**++*+*+*+*++*++*+$ |
| B | @+++*!*!*!+++**++++++++++++++++++*!*!*!*!*!*! |
| C | @++++++++++*++++*!+*++++++++++*!*!*!*!*!*$ |
| D | @*!++++++*!*!*!**!*!(6)+++++!*+++++++++*!*+**$ |

TABLE I
Synthesized programming patterns of different students solving an exercise

as student A and struggles with the errors in their program. Whereas student C on the other hand, manages to solve the exercise after after a series of errors. What knowledge or ability does student C have that student B fails to demonstrate. To get more insight into this we would have to look at the code snapshots from their exercises. Student D takes a six minute break after encountering several errors, and then goes on to finish the exercise. This could indicate that student D takes a strategic break after encountering several errors and comes back from the break with enough stamina to complete the exercise.

From these examples we can identify sequences of possible debugging, highlighted in pink. The frequency and length of these debugging sequences would be interesting to track over time and see how they change. Another applications could be to detect cheating among students.

This type of analysis could be automated for an entire class with dozens of assignments over weeks or months of instruction. Hopefully, if we collect this data we would be able to see what strategies and behaviors students use when programming and how these change over time.

## V. Conclusion

We are looking for feedback on this project idea. Specifically we are interested in hearing opinions about what are the most interested things to measure and what kinds of questions we would like to be able to answer with this type of analysis.

## References

[1] G. Gao, S. Marwan, and T. W. Price, "Early performance prediction using interpretable patterns in programming process data," in *Proceedings of the 52nd ACM technical symposium on computer science education*, 2021, pp. 342–348.

[2] J. Leinonen, F. E. V. Castro, A. Hellas *et al.*, "Fine-grained versus coarse-grained data for estimating time-on-task in learning programming," in *Proceedings of The 14th International Conference on Educational Data Mining (EDM 2021)*. The International Educational Data Mining Society, 2021.

[3] M. C. Jadud, "Methods and tools for exploring novice compilation behaviour," in *Proceedings of the second international workshop on Computing education research*, 2006, pp. 73–84.

[4] A. Alaboudi and T. D. LaToza, "Edit-run behavior in programming and debugging," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2021, pp. 1–10.